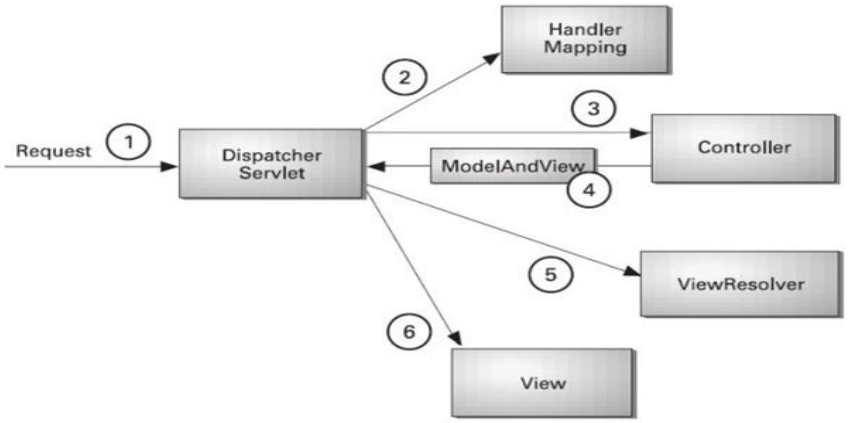
1. **Explain about Spring MVC?**

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

Model - A model contains the data of the application. A data can be a single object or a collection of objects.
Controller - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
View - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
Front Controller - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

2. **Explain Spring MVC flow?**



- ✓ After receiving an HTTP request, DispatcherServlet consults the Handler Mapping to call the appropriate Controller.
- ✓ The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.
- ✓ The DispatcherServlet will take help from View Resolver to pick up the defined view for the request.
- ✓ Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

3. **Explain the advantages of Spring MVC?**

- ✓ **Light Weight:** It uses light-weight servlet container to develop and deploy your application.
- ✓ **Speed development:** As we are suing the layered architecture it's easy to develop the code in faster manner
- ✓ **Separate roles:** The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
- ✓ **Reusable business code:** Instead of creating new objects, it allows us to use the existing business objects.
- ✓ **Easy to test:** In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter method.

4. Explain the disadvantages of Spring MVC?

- ✓ Complexity
- ✓ Not suitable for smaller applications.
- ✓ Bigger learning curve
- ✓ Lot of configurations
- ✓ Lot of xml files.

5. Explain about the Model Interface and its methods?

In Spring MVC, the model works a container that contains the data of the application. Here, a data can be in any form such as objects, strings, information from the database, etc.

It is required to place the Model interface in the controller part of the application. The object of HttpServletRequest reads the information provided by the user and pass it to the Model interface. Now, a view page easily accesses the data from the model part.

Methods:
==========
addAllAttributes(Collection<?> arg): It adds all the attributes in the provided Collection into this Map.
mergeAttributes(Map<String,?> arg): It adds all attributes in the provided Map into this Map, with existing objects of the same name taking precedence.
boolean containsAttribute(String arg): It indicates whether this model contains an attribute of the given name

6. Explain different types of view resolvers in Spring MVC?

- ✓ FreeMarkerViewResolver (Rendering the FTL)
- ✓ InternalResourceViewResolver (Rendering the JSP)
- ✓ XmlViewResolver (Rendering the XML files)

- ✓ UrlBasedViewResolver (Rendering based on URL)

7. Explain few annotations you have used in Spring MVC?

@Required: This annotation is applied on bean setter methods. Consider a scenario where you need to enforce a required property. The @Required annotation indicates that the affected bean must be populated at configuration time with the required property. Otherwise an exception of type BeanInitializationException is thrown.

```
public class RequestBean {

    int id;

    @Required
    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
```

@Autowired: This annotation is applied on fields, setter methods, and constructors. The @Autowired annotation injects object dependency implicitly.

```
@Autowired //Autowired byName
private AddressBean addressBean;

@Autowired ////Autowired constuctor
public StudentBean(int id, String name,List<String> address,AddressBean addressBean) {
    this.id=id;
    this.name=name;
    this.address=address;
    this.addressBean=addressBean;
}

@Autowired //Autowire byType
public void setId(int id) {
    this.id = id;
}
```

@Configuration: It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

```
@Configuration
public class SessionBean {

    @Bean
    public ProtoTypeBean getProtoTypeBean() {
        return new ProtoTypeBean();
    }
```

@ComponentScan: It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

```
@Configuration
@ComponentScan(basePackages = "com.example.demo")
public class SessionBean {
```

@Bean: It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

```
public class SessionBean {

    @Bean
    public ProtoTypeBean getProtoTypeBean() {
        return new ProtoTypeBean();
    }
```

@Value: The @Value annotation o inject values from a property file into a bean's attribute.

```
@Value("${enable.button}")
private String name;
```

@Component: It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with @Component is found during the classpath. The Spring Framework pick it up and configure it in the application context as a Spring Bean.

```
@Component
public class StudentBean {
```

@Service: It is also used at class level. It tells the spring that class contains the business logic. It will perform calculations and call external APIs

```
@Service
public class StudentBean {
```

@Repository: It is a class-level annotation. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.

```
@Repository
public class StudentBean {
```

@Qualifier: This annotation is used to avoid confusion which occurs when you create more than one bean of the same type and want to wire only one of them with a property. This annotation

is used along with @Autowired annotation. If more than one bean of the same type is available in the container, the framework will throw NoUniqueBeanDefinitionException.

```
Lets take an example. I have Vehicle base interface it has a method start().

public interface Vehicle {
public void start();
}

Here Care class imlementing the Vehicle interface and providing the implementation.

@Component
public class Car implements Vehicle {
   public void start() {
      System.out.println("Car started");
   }
}

Here Bike class imlementing the Vehicle interface and providing the implementation.

@Component
public class Bike implements Vehicle {
   public void start() {
      System.out.println("Bike started");
   }
}

Created one service class to executing the start method of the Vehicle interface,

@Component
public class VehicleService {
   @Autowired
   private Vehicle vehicle;
   public void service() {
      vehicle.start();
}
As there are two candidates (Car and Bike) for Vehicle annotated with @Autowire in the VehicleService bean.
The Spring IoC container unable to determine which bean should be injected and
throws an exception NoUniqueBeanDefinitionException.

To avoid the above problem @Qualifier will be useful,

Adding the unique bean definition for Car bean with @Qualifier

@Component
@Qualifier("carBean")
public class Car implements Vehicle {
public void start() {
System.out.println("Bike started");
   }
}

Adding the unique bean definition for Bike bean with @Qualifier

@Component
@Qualifier("bikeBean")
public class BIke implements Vehicle {
public void start() {
System.out.println("Bike started");
   }
}
```

```
Invoking the car class methods.

@Component
public class VehicleService {

   @Autowired
   @Qualifier("carBean")
   private Vehicle vehicle;

   public void service() {
      vehicle.start();
}
```

@Controller: The @Controller annotation is used to indicate the class is a spring controller. This annotation can be used to identify controllers for Spring MVC.

@RequestMapping: This annotation is used both at class and method level. It is used to map the web requests.

@ResponseBody: It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.

@RequestBody: This annotation is used to annotate request handler method arguments. The @RequestBody annotation indicates that a method parameter should be bound to the value of the HTTP request body. The HttpMessageConveter is responsible for converting from the HTTP request message to object.

8. **How to handle the validation errors in spring MVC?**

BindingResult holds the result of a validation and binding and contains errors that may have occurred. The BindingResult must come right after the model object that is validated or else spring fails to validate the object and throws an exception.

```
@NotBlank
@Size(min=2)
private String name;

@NotBlank
@Email
private String email;
```

```
@PostMapping("/")
public String checkForm(@Valid UserForm userForm, BindingResult bindingResult,
                        RedirectAttributes atts) {

    if (bindingResult.hasErrors()) {
        return "form";
    }
```

During the program execution as we mentioned the @Valid annotation before the Bean and added @NotBlank and other annotations Binding result will hold the errors. We can display the errors in UI like below,

```
<form action="#" class="ui form" th:action="@{/}" th:object="${userForm}" method="post">
    <div class="field">
        <label>Name:</label>
        <input type="text" th:field="*{name}">
        <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}">Name Error</span>
    </div>
```

9. Explain the steps to build an XML based Spring MVC application?

- ✓ Load the spring jar files or add dependencies in the case of Maven.
- ✓ Define the Controller.
- ✓ Create the view components like JSP/HTML/FTL.
- ✓ Provide the entry of FrontController in the web.xml file

```
<servlet>
 <servlet-name>spring</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>
```

- ✓ Define the Springs configuration xml file and provide the component scan and annotation driven tags along with view resolver information,

```
<!-- Provide support for component scanning -->
<context:component-scan base-package="com.javatpoint" />

<!--Provide support for conversion, formatting and validation -->
<mvc:annotation-driven/>
<!-- Define Spring MVC view resolver -->
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"></property>
    <property name="suffix" value=".jsp"></property>
 </bean>
</beans>
```

10. Explain the steps to build an Annotation based Spring MVC application?

- ✓ Load the spring jar files or add dependencies in the case of Maven.

- ✓ Define the Controller.
- ✓ Create the view components like JSP/HTML/FTL.
- ✓ For registering the DispatcherServlet instead of web.xml for programming Implement the interface WebApplicationInitializer and override its method onStartup()

```
public class AppInitializer implements WebApplicationInitializer {

        public void onStartup(ServletContext container) throws ServletException {

                AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContex
t();
                ctx.register(WebConfig.class);
                ctx.setServletContext(container);
                ServletRegistration.Dynamic servlet = container.addServlet("dispatcher", new Dispatc
herServlet(ctx));
                servlet.setLoadOnStartup(1);
                servlet.addMapping("/");
```

- ✓ Instead of spring xml configuration file we need to create a configuration class that should extend WebMvcConfigurerAdapter. Along with it we need to add the below annotations,

  @EnableWebMvc: For enabling annotation based configuration.
  @Configuration: Generates bean definitions
  @ComponentScan: Read the base packages classes.

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.devglan")
public class WebConfig extends WebMvcConfigurerAdapter {

        @Bean
        public ViewResolver viewResolver() {
                InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
                viewResolver.setViewClass(JstlView.class);
                viewResolver.setPrefix("/WEB-INF/jsp/");
                viewResolver.setSuffix(".jsp");
                return viewResolver;
        }

        @Override
        public void addResourceHandlers(final ResourceHandlerRegistry registry) {
                registry.addResourceHandler("/js/**").addResourceLocations("/ui/js/");
                registry.addResourceHandler("/css/**").addResourceLocations("/ui/css/");
        }
```

11. **Given the steps to integrate the Jpa with Spring MVC App?**

- ✓ Load the spring jar files or add dependencies in the case of Maven.
- ✓ Define the Controller.
- ✓ Create the view components like JSP/HTML/FTL.

- ✓ For registering the DispatcherServlet instead of web.xml for programming Implement the interface WebApplicationInitializer and override its method onStartup()

```
public class AppInitializer implements WebApplicationInitializer {

        public void onStartup(ServletContext container) throws ServletException {

                AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContex
t();
                ctx.register(WebConfig.class);
                ctx.setServletContext(container);
                ServletRegistration.Dynamic servlet = container.addServlet("dispatcher", new Dispatc
herServlet(ctx));
                servlet.setLoadOnStartup(1);
                servlet.addMapping("/");
```

- ✓ Instead of spring xml configuration file we need to create a configuration class that should extend WebMvcConfigurerAdapter. Along with it we need to add the below annotations,

  @EnableWebMvc: For enabling annotation based configuration.
  @Configuration: Generates bean definitions
  @ComponentScan: Read the base packages classes.

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.devglan")
public class WebConfig extends WebMvcConfigurerAdapter {

        @Bean
        public ViewResolver viewResolver() {
                InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
                viewResolver.setViewClass(JstlView.class);
                viewResolver.setPrefix("/WEB-INF/jsp/");
                viewResolver.setSuffix(".jsp");
                return viewResolver;
        }

        @Override
        public void addResourceHandlers(final ResourceHandlerRegistry registry) {
                registry.addResourceHandler("/js/**").addResourceLocations("/ui/js/");
                registry.addResourceHandler("/css/**").addResourceLocations("/ui/css/");
        }
```

- ✓ Create the persistence.xml file with all the database details,

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
          http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
    version="2.1">

    <persistence-unit name="SalesDB">
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/sales" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="password" />
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
        </properties>
    </persistence-unit>

</persistence>
```

- ✓ Create a configuration class responsible for reading the database details from persistence.xml and it will return the Transaction Manager class. For JPA add the below annotations on it,

  @Configuration: Generate the beans definitions
  @EnableJpaRepositories: It will load the Jpa repository classes.
  @EnableTransactionManagement: Enabling the Jpa related annotations.

```java
@Configuration
@EnableJpaRepositories(basePackages = {"net.codejava.customer"})
@EnableTransactionManagement
public class JpaConfig {
    @Bean
    public LocalEntityManagerFactoryBean entityManagerFactory() {
        LocalEntityManagerFactoryBean factoryBean = new LocalEntityManagerFactoryBean();
        factoryBean.setPersistenceUnitName("SalesDB");

        return factoryBean;
    }

    @Bean
    public JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
        JpaTransactionManager transactionManager = new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(entityManagerFactory);

        return transactionManager;
    }
```

12. Is Singleton beans are thread safe in Spring Framework?

No, singleton beans are not thread-safe in spring framework.

13. Explain the Difference Between @Controller and @RestController?

The main difference between the @Controller and @RestController annotations is that the @ResponseBody annotation is automatically included in the @RestController. This means that we don't need to annotate our handler methods with the @ResponseBody. We need to do this in a @Controller class if we want to write response type directly to the HTTP response body.

14. Explain Model, ModelMap and ModelAndView?

- ✓ The Model interface defines a holder for model attributes.
- ✓ The ModelMap has a similar purpose, with the ability to pass a collection of values.
- ✓ The ModelAndView, we return the object itself. We set all the required information, like the data and the view name, in the object we're returning.

15. Explain SessionAttributes and SessionAttribute?

When developing web applications, we often need to refer to the same attributes in several views. For example, we may have shopping cart contents that need to be displayed on multiple pages. A good location to store those attributes is in the user's session. We use it at the controller class level.

```
@Controller
@RequestMapping("/sessionattributes")
@SessionAttributes("todos")
public class TodoControllerWithSessionAttributes
```

If we want to retrieve the existing attribute from a session that is managed globally, we'll use @SessionAttribute annotation as a method parameter:

```
@GetMapping
public String getTodos(@SessionAttribute("todos") TodoList todos) {
    // method body
    return "todoView";
}
```

16. What Is Spring MVC Interceptor and How to Use It?

Spring MVC Interceptors allow us to intercept a client request and process it at three places – before handling, after handling, or after completion (when the view is rendered) of a request.

The interceptor can be used for cross-cutting concerns and to avoid repetitive handler code like logging, changing globally used parameters in Spring model, etc.

17. What is the difference between @ComponentScans and @ComponentScan?

@ComponentScan: configures which packages to scan for classes with annotation configuration. We can specify the base package names directly with one of the basePackages or value arguments (value is an alias for basePackages)

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
class VehicleFactoryConfig {}
```

Also, we can point to classes in the base packages with the `basePackageClasses` argument:

```
@Configuration
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

@ComponentScan leverages the Java 8 repeating annotations feature, which means we can mark a class with it multiple times:

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

@ComponentScans to specify multiple @ComponentScan configurations:

```
@Configuration
@ComponentScans({
    @ComponentScan(basePackages = "com.baeldung.annotations"),
    @ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
})
class VehicleFactoryConfig {}
```

18. **Difference between basePackages and basePackageClasses in @ComponentScan?**

    basePackages: Which is used to read the classes under the package.

    basePackageClasses: Which is used to read the classes.

19. **Can we have multiple spring configuration files?**

    YES. You can have multiple spring context files. There are two ways to make spring read and configure them.

    Specify all files in web.xml file using contextConfigLocation init parameter

```
<servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
                WEB-INF/spring-dao-hibernate.xml,
                WEB-INF/spring-services.xml,
                WEB-INF/spring-security.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

OR, you can import them into existing configuration file you have already configured

```
<beans>
    <import resource="spring-dao-hibernate.xml"/>
    <import resource="spring-services.xml"/>
    <import resource="spring-security.xml"/>

    ... //Other configuration stuff

</beans>
```

20. What is DispatcherServlet and ContextLoaderListener?

DispatcherServlet: It will use the handler mapping inside and render the request to specific controller.

ContextLoaderListener: reads the spring configuration file (with value given against “contextConfigLocation” in web.xml), parse it and loads the beans defined in that config file.

```
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>

    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>
</servlet>
```